Rapport du PSAR/PSTL : Parallélisation de graphes audio (sujet PSTL)

Eisha Chen-yen-su, Ivan Delgado

26 mai 2019

Introduction

Le présent document porte sur un projet STL, il a donc été réalisé dans le cadre de ce dernier mais également dans celui du projet SAR car l'un des membres du binôme est en SAR. Il comportera donc des éléments provenant du cahier des charges qui a été rendu pour le PSAR, mais comme ce dernier n'était pas exigé pour le PSTL, il nous a semblé judicieux de faire cela pour qu'il ne manque aucun éléments de contexte.

Dans ce rapport, nous aborderons en premier lieu les graphes audio, ainsi que ce qui les caractérisent. Nous présenterons ensuite les algorithmes d'ordonnancement ainsi que l'architecture logicielle ayant permis leur mise en œuvre. Nous ferons également une comparaison entre les performances de ces ordonnancements. Finalement, nous ferons une critique de la réalisation du projet en indiquant les aspects de l'implémentation qui pourraient être améliorés et les fonctionnalités qui pourraient être développées par la suite.

Présentation des besoins

Les systèmes interactifs musicaux (abrégé SIM) permettent de jouer et de composer de la musique en temps réel. Ces derniers permettent la synthèse de signaux audio, qui est le résultat d'un ensemble de traitements sur ces signaux. Ces traitements doivent se faire en temps réel : à chaque cycle audio, tous les calculs doivent se faire dans un temps imparti (i.e. avant la fin du cycle), sans quoi, il y a une dégradation de la qualité du signal audio. Or la complexification de ces traitements font que cette synthèse peut nécessiter de plus en plus de calculs mais tout en devant être sur les mêmes durées, il devient donc difficile de ne pas avoir de dégradations dans ces conditions.
Une solution consiste à tirer profit de l'architecture multi-cœur des processeurs modernes pour paralléliser ces calculs.

Nous allons préciser ici les contraintes temps réel que doivent respecter les traitements audio en nous inspirant de l'analyse réalisée dans [1].
Un signal audio est caractérisé par sa fréquence d'échantillonnage indiquant le nombre de valeurs (chaque valeur étant un nombre à virgule flottante ou un entier) qu'il doit prendre à chaque secondes, le plus souvent elle est de 44,1 kHz. La carte son d'un ordinateur va lire ces échantillons dans un buffer audio de taille fixe à un intervalle régulier (c'est ce qui défini un cycle audio), donc si nous avons un buffer de 512 échantillons, la carte va lire dans ce buffer environ 86 fois par seconde (44 100 divisé par 512), donc toutes les 11,6 millisecondes [1]. Si un nouveau buffer n'est pas disponible au delà de cette échéance, alors la carte audio va se mettre à lire des échantillons nuls ou bien le même buffer qu'au cycle précédent, ce qui produit un "tick" désagréable à l'oreille. La durée d'un cycle audio constitue donc la contrainte temps réel que doit

1. L'API JACK que nous utilisons utilise deux buffers, le principe est le même, la durée des cycle est doublée

respecter le traitement audio : le temps de calcul des traitements audio, pour chaque buffer, doit impérativement être inférieur à la durée d'un cycle.

Ces traitements audio peuvent êtres représentés par un graphe audio (ou DAG, pour "Directed Acyclic Graph"). Les nœuds d'entrées d'un tel graphe sont les sources des signaux audio et les nœuds de sorties, les sorties audio du système. Les autres nœuds du graphe sont des traitements altérant les signaux audio passant par ces derniers. Ainsi chaque signal va-t-il suivre un chemin audio en passant d'un nœud à un autre jusqu'à arriver à l'une des sorties.

Un graphe audio peut être vu comme un graphe de tâches dont chaque nœud représente un traitement audio (en anglais, "DSP" pour "Digital Signal Processor") et les arcs représentent les buffers ou canaux permettant à deux DSP de communiquer entre eux. De plus, il est évident qu'un arc représentant une communication d'un nœud A vers B induit que la tâche B dépend de la tâche A qui doit s'exécuter avant.

Le poids d'un nœud correspond au temps nécessaire à l'exécution d'une tâche. Celui d'un arc correspond au coût de communication entre deux tâches. Le coût de communication peut être considéré comme négligeable dans le cas d'une communication via une mémoire partagée, elle peut être beaucoup plus importante et variable dans le cas de passages de messages, il faut alors en tenir compte.

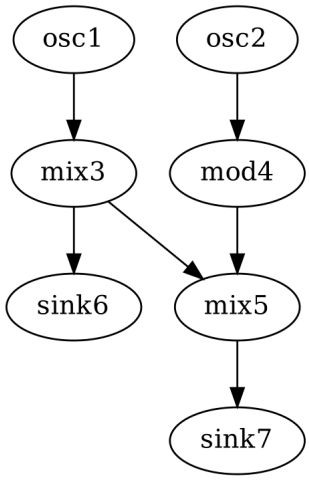


FIGURE 1 – Exemple de DAG

Le chemin critique d'un graphe de tâches correspond au plus long chemin (par rapport à la somme totale des poids des nœuds et des arcs le constituant) entre un nœud d'entrée et un nœud de sortie du DAG. Et la longueur de ce chemin est le temps minimum (possible) d'exécution parallèle de ce graphe.

Donc paralléliser un graphe de tâches revient à faire un ordonnancement périodique de ses tâches entre les différents processeurs disponibles, tout en respectant les dépendances entre ces dernières.

La parallélisation des graphes audio est déjà quelque chose de connue, cependant elle n'est pas supportée par tous les SIM ou alors elle exige l'utilisation d'instructions explicites. Notre objectif était donc d'étudier les performances d'algorithmes d'ordonnancement de graphes de tâches vis-à-vis des contraintes temps réel du domaine audio, pour permettre la parallélisation automatique de l'exécution de graphes audio.

Algorithmes utilisés

Nous allons à présent parler des différents algorithmes et stratégies utilisées pour étudier les possibilités de parallélisation automatique de graphes audio. Nous avons vu qu'il s'agissait d'un problème d'ordonnancement de graphes de tâches.

Taxonomie de l'ordonnancement

Tout d'abord, il y a deux types d'ordonnancements de DAG : l'ordonnancement statique, qui définit quelles tâches vont s'exécuter sur quels processeurs à un instant donné, avant son exécution (comme au moment de la compilation par exemple) et qui sera toujours le même ; l'ordonnancement dynamique qui quand à lui est déterminé au fur et à mesure de l'exécution du graphe et qui par conséquent, peut varier en fonction d'éventuels aléas des différentes exécutions.

Les algorithmes d'ordonnancement statique sont tirés de [1] et la stratégie d'ordonnancement dynamique est tirée de [2] et [3].

Nous pouvons dégager certaines caractéristiques des DAG dans les cas étudiés : il n'y a pas de tâches parallèles (i.e. une tâche n'est exécutée que sur un seul processeur), le coût de calcul des tâches est arbitraire et les coûts de communication entre les tâches seront considérés comme négligeables.

Ces DAG seront exécutés sur des systèmes multiprocesseurs à mémoire partagée et les processeurs ont tous la même vitesse de traitement.

Les algorithmes d'ordonnancement statique

Le principe de l'ordonnancement à liste

Nous allons définir le principe général derrière les algorithmes d'ordonnancement statiques.

Ces algorithmes construisent d'abord une liste des tâches ordonnées par ordre décroissant de priorité. À chaque étape, il retire la tâche la plus prioritaire de la liste, puis l'ordonnance sur le processeur permettant à la tâche de commencer le plus tôt possible. Il se termine lorsque toutes les tâches ont été assignées à l'un des processeurs. Certains algorithmes calculent les priorités des tâches une seule fois au début, d'autres les évaluent à chaque itération.

Deux attributs sont fréquemment utilisés pour déterminer la priorité d'une tâche : le t-level ("top level") et le b-level ("bottom level"). Le t-level d'un nœud n correspond à la longueur (i.e. la somme des poids des nœuds et des arcs) maximale d'un chemin allant de l'un des nœuds d'entrée du DAG vers n (en excluant son propre poids). Le b-level d'un nœud n est quand à lui la longueur maximale d'un chemin allant de n vers un nœud de sortie du graphe. Il y a aussi une variante statique du b-level pour laquelle nous faisons seulement la somme des poids des nœuds : le "static level" (que nous abrégeons "SL").

L'algorithme HLFET avec départage CP/MISF

Le premier algorithme est l'algorithme HLFET (pour « Highest Level First with Estimated Times ») avec un départage des égalités avec CP/MISF (pour « Critical Path / Most Immediate Successors First »). Pour celui-ci, la priorité d'un nœud est définie par son SL. Lorsque plusieurs nœuds ont le même SL, on choisi d'abord celui ayant le plus grand nombre de successeurs.

Lorsque l'on considère négligeable les coûts de communication entre les tâches, cet algorithme est supposé être proche de l'ordonnancement optimal, comme il privilégie les nœuds appartenant au chemin critique. De plus, il est très simple à mettre en place. C'est pour ces raisons que nous avons choisi cet algorithme en premier.

L'algorithme ETF

Le second algorithme statique est l'algorithme ETF (pour "Earliest Time First"). À chaque itération, il faut calculer pour chaque nœud prêt (i.e. dont tous les parents ont déjà été ordonnancés), et sur chaque processeur, l'instant le plus tôt auquel il pourra s'exécuter. Puis on ordonnance le nœud pouvant s'exécuter au plus tôt sur le processeur le permettant. Les égalités sont résolues en choisissant d'abord le nœud ayant le SL le plus élevé.

Cet algorithme est également simple à implémenter. De plus, il privilégie au mieux les temps de démarrage les plus précoces ainsi que les nœuds du chemin critique. Il est aussi possible de borner la qualité du résultat par rapport à l'ordonnancement optimal.

L'algorithme CPFD

Le dernier algorithme statique est l'algorithme CPFD (pour "Critical Path Fast Duplication"). Il est différent des autres vus précédemment, car il va ordonnancer les tâches via un procédé plus complexe qu'avec une liste ordonnée. CPFD s'autorise à dupliquer des tâches lorsque cela permet d'éviter les coûts de communication d'une tâche A vers une tâche B : si une tâche B s'exécute juste après A sur le même processeur, alors le résultat de A est déjà disponible (sur ce processeur) pour que B puisse s'exécuter immédiatement après. Il n'y a donc pas besoin d'envoyer de messages pour communiquer le résultat de A à B (ce qui peut être significativement long). Cela peut être particulièrement utile dans un système réparti : faire en sorte que deux tâches directement dépendantes s'exécutent sur un même site permet d'éviter des envoies de messages et peut donc faire gagner du temps.

Dans le cadre du domaine audio, cela peut être mis en application lors d'une collaboration en temps réel entre plusieurs musiciens étant sur plusieurs machines différentes ou éloignés géographiquement.

CPFD permet donc de prioriser les nœuds du chemin critique et peut faire plus de réductions de coûts de communication via la duplication de tâche, il semble donc bien adapté pour s'assurer de respecter au mieux une contrainte temps réel sur un système réparti.

Ce qui va suivre est la description de l'algorithme.

Premièrement, CPFD va distinguer les nœuds du DAG en trois catégories : les CPN (pour "Critical Path Node"), qui sont les nœuds appartenant à un chemin critique ; les IBN (pour "In-Branch Nodes"), qui sont les nœuds possédant un chemin menant à un CPN ; et les OBN (pour "Out-Branch Node"), qui sont simplement les nœuds n'appartenant pas aux deux autres catégories. CPFD s'appuie sur une liste appelée "CPN-Dominant Sequence". Elle est construite selon l'algorithme **??** page suivante :

Premièrement, CPFD va distinguer les nœuds du DAG en trois catégories :
— Les CPN (pour "Critical Path Node"), qui sont les nœuds appartenant à un chemin critique.
— Les IBN (pour "In-Branch Nodes"), qui sont les nœuds possédant un chemin menant à un CPN.
— Les OBN (pour "Out-Branch Node"), qui sont simplement les nœuds n'appartenant pas aux deux autres catégories.

Insérer en premier le CPN d'entrée du DAG dans la séquence, mettre *Position* à 2. Soit nx, le nœud suivant dans le chemin critique
Répéter :
    Si nx a tous ses parents dans la séquence
        mettre nx à *Position* dans la séquence et incrémenter *Position*.
    Sinon
        Posons ny tel qu'il soit un éventuel parent de nx qui ne soit pas dans la séquence et avec le *b-level* le plus élevé (si plusieurs nœuds correspondent à cela, on choisi d'abord celui ayant le t-level minimal).
        Si ny a tous ses parents dans la séquence
            alors l'insérer à *Position* et incrémenter *Position*.
        Sinon
            inclure récursivement tous les prédécesseurs de ny dans la séquence.
            Répéter l'étape précédente jusqu'à ce que tous les parents de nx soient dans la séquence. Insérer nx dans la séquence à *Position*.
        Fin si.
    Fin si.
Jusqu'à ce que tous les CPN soient dans la séquence.
Ajouter à la fin de la séquence, tous les OBN par ordre décroissant de *b-level*.

Algorithme 1 : Sequencement des CPN

À partir de cette séquence, CPFD procède à l'ordonnancement selons l'algorithme **??** page 24 :

La toute dernière stratégie d'ordonnancement étudiée est l'ordonnancement avec vole de tâches. Il s'agit d'un ordonnancement dynamique dans lequel les tâches sont exécutées par un pool de threads.

Chaque thread a une file d'attente. Au début de l'exécution du DAG, les tâches prêtes (i.e. celles correspondant aux nœuds d'entrée du DAG) sont réparties parmi les files d'attentes de chacun des threads en étant insérées au début de ces files. Pour avoir la prochaine tâche à exécuter, un thread en prend une au début de sa file (ordre LIFO). Si sa file est vide, il va "voler" une tâche à la fin (ordre FIFO) de la file d'attente d'un autre thread.

À chaque fois qu'un thread fini une tâche, il va ajouter, au début de sa file, la (ou les) tâche(s) nouvellement prête(s) (i.e. qui en dépendait et dont leurs autres dépendances ont également été satisfaites).

Cette stratégie a pour premier avantage d'utiliser des files d'attentes qui peuvent êtres "lock-free", donc il y a peu de contentions sur ces dernières et il n'y a pas de surcoûts causés par des synchronisations. Comme un thread va toujours tenter de suivre un chemin de calcul du DAG (car il va essayer d'exécuter immédiatement, après une tâche, ses successeurs), il y a augmentation de la localité des données et on a alors une plus grande probabilité pour que les données, sur lesquelles travaille un thread, restent dans les caches du processeur. De plus, lorsqu'un thread vole une tâche dans une file F, cette tâche sera celle avec le *t-level* le plus petit de la file F, donc avec une priorité plus élevée que les autres.

Architecture logicielle

Dans cette partie, nous ne parlerons pas de façon détaillée de l'implémentation : en effet, cette dernière est déjà amplement commentée et documentée dans le code source du projet. De même, la documentation utilisateur est contenue dans le README du projet [Dépôt : https ://gitlab.com/Vonstrab/audio-graph-parallelization].

Nous nous concentrerons ici sur la présentation de l'architecture globale ainsi que sur les points importants de l'implémentation.

Les fonctionnalités réalisées

Nous allons présenter les fonctionnalités réalisées par le logiciel développé durant le projet. Le logiciel peut lire certains graphes audio écrits dans des fichiers au format AudioGraph. Il peut créer des fichiers DOT et PDF (avec **Graphviz** [4]) représentant le graphe. Il peut calculer des ordonnancements statiques pour ces graphes avec les algorithmes *HLFET*, *ETF*, *CPFD* ou "random" (on assigne des priorités aléatoires aux nœud lors de l'ordonnancement) et éventuellement les afficher. Il peut exécuter séquentiellement le graphe audio ou en parallèle avec des threads **POSIX** selon un ordonnancement statique ou un ordonnancement dynamique avec vol de tâches. Pour finir, il est également possible de faire des mesures sur les temps des exécutions d'un graphe audio.

Les technologies utilisées

Nous allons à présent parler des technologies qui sont utilisées par ce projet.

Le logiciel est écrit en **Rust** [5]. C'est un langage de programmation dont le compilateur produit du code rapide (le compilateur utilisant **LLVM** [6]comme backend), il est orienté vers la programmation concurrente et le compilateur fait de nombreuses vérifications pour détecter des problèmes liés à la mémoire (comme par exemple le déréférencement de pointeurs non valides ou les data races).

Jack [7] est utilisé pour la fonction de callback audio. **JACK** fait référence à une API et aussi à l'implémentation d'une infrastructure permettant à des applications audio de communiquer entre elles et avec les interfaces audio (comme des cartes son). À chaque cycle audio, le serveur audio **JACK** va appeler la fonction de callback audio de notre application. C'est dans cette fonction que va être appelée notre routine d'exécution du graphe audio. Les nœuds de sortie du graphe vont écrire les résultats des traitements du graph dans les buffers des ports de sortie de l'application, ce sont ces buffers qui vont être lus par le serveur **JACK**.

crossbeam [8] est utilisé pour diverses mécanismes de synchronisation tels que :

- Les *channel* qui sont des canaux de messages multi-producteurs et multi-consommateurs.
- Les *deque* qui sont une implémentation de files utilisables pour l'ordonnancement par vole de tâche.
- Les *ShardedLock* qui sont des verrous permettant à une ressource d'être verrouillée pour un seul écrivain ou pour plusieurs lecteurs. Ce type de verrous existent déjà dans la bibliothèque standard (*RwLock*) mais ceux de **crossbeam** sont plus rapide pour acquérir le verrou en lecture mais plus lent pour l'acquérir en écriture, ce qui est mieux lorsque l'on sait qu'on va plus souvent lire une donnée que la modifier.
- L'utilitaire *Backoff* permet de faire des boucles d'attentes actives mais en réduisant la contention sur le processeur, en faisant en sorte que le thread rende la main à l'OS, au bout d'un certain temps, pour des durées qui croissent exponentiellement à chaque fois.

core_affinity [9] permet de faire en sorte qu'un thread s'exécute toujours le même processeur. Ceci est extrêmement important pour la performance de l'exécution des DAG, car les algorithmes d'ordonnancement s'appuient sur la localité des données : lorsque deux tâches s'exécutent successivement et que l'une utilise le résultat de l'autre (i.e. il y a une relation de dépendance entre ces tâches dans le DAG), il est beaucoup plus avantageux qu'elles s'exécutent sur le même cœur car la seconde tâche peut plus rapidement accéder aux données qui ont été écrites par la première et qui sont encore dans le cache du cœur, au lieu d'avoir à les chercher dans la RAM, ce qui est systématiquement le cas si les tâches s'exécutent sur des cœurs différents.

criterion [10] permet de faire des benchmarks pour estimer les WCET (Worst Case Execution Time) des nœuds du DAG, qui seront utilisés comme coût de calcul des nœuds par les algorithmes d'ordonnancement statiques.

pest [11] est utilisé pour écrire le parser de fichiers **AudioGraph**.

Présentation des modules Rust

Rust permet de simplement structurer le code d'un logiciel en modules.
Voici la liste des modules du projet :

- *dsp* : contient l'implémentation des traitements audio ainsi que des buffers permettant de communiquer entre les tâches.
- *execution* : implémente les exécutions des DAG. Il y a l'exécution séquentielle mais aussi l'exécution parallèle des ordonnancements statiques et l'exécution parallèle avec ordonnancement par vol de tâches.
- *measure* : contient les fonctions permettant d'effectuer les mesures temporelles sur les différentes exécutions.
- *parser* : contient le parser pour extraire les graphes audio décrits dans des fichiers AudioGraph.
- *static_scheduling* : contient l'implémentation des algorithmes d'ordonnancement statique.
- *task_graph* : implémente la représentation d'un DAG avec diverses informations utilisées par les algorithmes d'ordonnancement statique ou l'exécution parallèle avec ordonnancement par vol de tâches.

Les DSP et fichiers AudioGraph supportés

Nous allons détailler ici la manière dont a été implémenté le traitement du signal. Nous nous sommes efforcé de garder un séparation entre la représentation du DAG contenant les informations utiles à l'ordonnancement (le graphe de tâches) et le graphe audio sous-jacent où est effectué le traitement du signal.

Les nœuds de ce graphe audio sont des fonctions prenant un (ou plusieurs) signal en entrée, sous forme d'un (ou plusieurs) tableau de flottants (qui sont les échantillons audio), et écrivant le résultat du traitement dans un (ou plusieurs) autre tableau de flottants. Les nœuds d'entrée du graphe ne prennent pas de signal en entrée mais en fournissent un en sortie, les nœuds de sortie ont seulement une entrée.

Les arcs du graphes sont les buffers reliant les traitements : une fonction de traitement écrit son résultat dans ce buffer (c'est sa sortie) et une autre lit dans ce même buffer (c'est son entrée). Ces buffers viennent de la structure *DspEdge* qui contient toutes les caractéristiques du signal : les échantillons du signal, la fréquence d'échantillonnage et la taille du buffer.

Les DSP implémentés sont les suivants :

- *Oscillator* : un oscillateur produisant une onde sinusoïdale avec une certaine amplitude et fréquence. Il n'a qu'une seule sortie.
- *Modulator* : applique une modulation au signal d'entrée qui peut s'apparenter à de la modulation d'amplitude (AM). Il a exactement une entrée et une sortie.
- *Sink* : ce DSP est la sortie du graphe audio, il écrit le signal d'entrée dans un buffer audio de **JACK**. Il n'a qu'une seule entrée.
- *InputOutputAdaptator* : il s'agit d'un mixeur, il peut avoir plusieurs entrées et sorties. Il mixe les signaux en entrée et écrit le même résultat tous ses buffers de sortie.

Lorsque le programme lit un fichier AudioGraph, il va reconnaître les nœuds déclarés comme les DSP implémentés. Si le parser trouve un nœud dont le type est inconnu, il va lui assigner un DSP par défaut en fonction du nombre de ses entrées et sorties.

Implémentation de l'exécution séquentielle

L'exécution séquentielle du graphe audio est réalisée dans la fonction *run_seq*. Cette fonction va initialement faire une liste des tâches en effectuant un tri topologique du DAG. Puis, à chaque cycle audio, va exécuter le DAG en exécutant

séquentiellement les tâches dans l'ordre de cette liste.

Implémentation des algorithmes d'ordonnancement statique

Les ordonnancements statiques ont été implémentés avec des fonctions prenant en paramètre un graphe de tâche (la structure *TaskGraph*) et retournant un ordonnancement. L'ordonnancement est représenté dans la structure Schedule qui contient la liste des processeurs participant à l'exécution des tâches (la structure *Processor*). Chaque processeur a une liste de fenêtres temporelles (la structure *TimeSlot*) indiquant quand il doit commencer une tâche et quand il doit la finir (qui est simplement la date de début plus le WCET), ces fenêtres ne se chevauchent évidemment pas, car un même processeur ne peut qu'exécuter séquentiellement les tâches.

Il y a quatres fonctions pour ces ordonnancements : *random* qui réalise une stratégie d'ordonnancement où l'on donne une priorité aléatoire aux différents nœuds, et *hlfet*, *etf* et *cpfd* qui réalisent les algorithmes du même nom.

L'exécution des ordonnancements statiques est réalisée dans la fonction *run_static_sched*. Cette fonction va initialement calculer l'ordonnancement statique sur n processeurs (attention, ce nombre ne soit pas dépasser le nombre de cœurs disponibles sur la machine) du DAG en fonction d'un des algorithmes implémentés, puis crée un pool de n thread qui a connaissance de l'ordonnancement. Ainsi chaque thread se retrouve affecté à l'exécution de l'un des "processeurs" décrit dans l'ordonnancement statique et chaque thread va toujours s'exécuter sur un même cœur, grâce à **core_affinity**. On a alors une bonne correspondance entre ce qui est décrit dans les ordonnancements statique et ce qui sera réellement effectué lors des exécutions du DAG.

Chaque thread a donc une liste de tâche (celle de la structure Processor qui leur a été attribuée) et va l'exécuter une à une en respectant l'ordre. Cependant, cette implémentation ne tient pas compte des fenêtres temporelles de la liste de tâches mais va essayer d'exécuter les tâches dès que possible (car le plus souvent, le calcul d'une tâche prendra moins de temps que le WCET), mais il y a un mécanisme de synchronisation pour que les dépendances entre les tâches soient quand même respectées.

Chaque tâche est associée à un état (l'énumération *TaskState*) indiquant si elle est prête à être exécutée (i.e. si toutes les tâches parentes ont fini de s'exécuter) ou non. Dans le cas où elle ne l'est pas, il y a un compteur indiquant combien de parents doivent encore s'exécuter (le compteur d'activation).

Quand un thread finit d'exécuter une tâche, il change son état pour indiquer qu'elle est finie et va décrémenter les compteurs d'activation des tâches dépendantes et si le compteur d'activation d'une tâche atteint 0, elle devient prête.

Quand un thread veut commencer à exécuter une tâche, il doit d'abord vérifier si celle-ci est prête. Si ce n'est pas le cas, il entamer une attente active avec l'utilitaire *Backoff*. L'attente active permet d'éviter que les threads ne s'endorment systématiquement pour attendre qu'une tâche soit prête. Ce genre de mécanisme de synchronisation engendre des appels systèmes, ce qui d'une part utilise beaucoup de cycles CPU et d'une autre, comme les threads doivent rendre la main au noyau, lorsqu'ils reviennent de l'appel système, les données qui leur sont utiles ont probablement été évincées des caches du processeur. Pour ces raisons, il faut éviter le plus possible de faire des appels systèmes lorsqu'on programme une application temps réel.

Implémentation de l'exécution de l'ordonnancement dynamique avec vol de tâches

L'exécution parallèle avec l'ordonnancement par vol de tâches est réalisée dans la fonction run_work_stealing. La fonction va initialement créer un pool de n threads. Chaque thread a sa propre file de tâches (de type Worker, venant de crossbeam) et il y a une file globale (de type Injector). Les threads vont initialement chercher les tâches dans la file globale. Puis, à chaque fin de tâche, ils vont décrémenter les compteurs d'activations des tâches dépendantes (cela marche comme

pour l'exécution des ordonnancements statiques), ajouter au début de leur file les nouvelles tâches prêtes et finalement prendre la prochaine tâche, au début de leur file ou en volant à la fin d'une des autres files.

Mécanisme de contrôle des pools de threads

Pour les deux exécutions parallèles, lorsque le programme doit commencer l'exécution d'un graphe, il appel une méthode du thread de pool qui envoie un message à chaque thread pour leur indiquer de commencer les calculs. Chaque thread dispose d'un canal de messages sur lequel il attend un message avant de commencer à exécuter le graphe. À chaque fois qu'un thread fini toutes les tâches à exécuter, il va attendre sur son canal le prochain message lui indiquant que le graphe doit à nouveau être exécuté.

Réalisation des mesures de performance

Nous voulions faire des mesures sur le temps d'exécution total d'un graphe à chaque cycle audio. Pour cela, au début de la fonction de callback audio, la date est enregistrée. Il y a un second enregistrement de la date juste après la fin de l'exécution du graphe audio. Au lancement du programme, il y a un thread qui est créé et qui est associé à un canal de messages. Ce thread va simplement attendre de recevoir des messages sur ce canal et va enregistrer dans un fichier le contenu de chaque message reçu. À chaque fois que le thread principal fait un enregistrement, il envoie un message contenant l'information enregistrée au thread qui va écrire ces informations dans un fichier. L'intérêt d'utiliser un thread séparé pour écrire les mesures dans un fichier est d'éviter aux autres threads d'avoir à effectuer des appels systèmes (ce qui doit être fait si l'on veut manipuler des fichiers), ils n'ont qu'à envoyer un message dans le canal vers le thread écrivain et cette opération est toujours non-bloquante pour les canaux de type unbounded (fournis par crossbeam). Les mesures ont ainsi un impact limité sur l'exécution des graphes audio.

Un script Python a été écrit pour permettre de facilement générer des données statistiques permettant de faire des comparaisons entre les différents algorithmes de parallélisation. Le script va lancer plusieurs fois différents programmes sur différents fichiers pendant quelques secondes. À l'issue de cette étape, il y a plusieurs fichiers contenant les informations sur les temps d'exécutions des graphes audio dans un répertoire. Le script va alors parcourir ces fichiers pour générer des graphiques présentant des statistiques sur les exécutions.

Comparaison des Algorithmes d'ordonnancement

Dans cette partie, nous allons réaliser une comparaison des performances des différents ordonnancements.

Les caractéristiques des systèmes exécutant ces graphes

L'architecture des machines, qui vont exécuter ces DAG, considérées dans le cadre de ce projet correspond aux machines utilisées dans la vie courante (i.e. ordinateurs de bureau et portables). Ce sont donc des systèmes à mémoire partagée dotés d'un nombre plutôt restreint de processeurs. Ce sont des systèmes homogènes car les processeurs ont tous la même vitesse.

De plus, les systèmes sur lesquels fonctionnent la plupart des SIM, ne sont pas des systèmes temps réel (les systèmes de la famille Unix (GNU/Linux ou MacOS) ou Windows, par exemple). Même si les noyaux de ces systèmes peuvent avoir une notion de processus avec une priorité temps réel, ils ne sont pas préemptifs et ne peuvent donc pas toujours garantir le temps de réponse à une interruption, ni avoir certaines autres garanties temps réel.

Ne pouvant pas avoir un contrôle total sur l'ordonnancement des threads/processus constituant les SIM (parfois, il n'est même pas possible d'avoir une priorité temps réel), il se peut que parfois le SIM ne puisse pas avoir assez de temps processeur à un certain moment pour respecter la contrainte temps réel. Il peut aussi y avoir une perte de performance par rapport à la gestion des caches des processeurs : par exemple, s'il y a une tâche B dépendant du résultat d'une tâche A, si A s'exécute sur un processeur et que B s'exécute ensuite sur un processeur différent, le résultat de A sera obligé de transiter par la mémoire centrale de la machine pour pouvoir être lue par B, ce qui prendrait plus de temps que si B pouvait ensuite s'exécuter sur le même processeur que A et que le résultat était resté dans le cache ; de manière similaire, si A et B vont s'exécuter sur un même processeur, s'il y a un autre processus (ou même un autre thread du SIM) qui s'exécute entre les deux, sur ce processeur, alors ce dernier peut provoquer l'éviction du résultat de A depuis le cache du processeur, avant que B n'ait pu s'en servir. Nous serons donc dans une démarche de "best-effort" en essayant de trouver les meilleures solutions vis-à-vis des limitations que ces systèmes nous imposent.

Dans cette optique regarder juste le temps moyen uniquement n'est pas pertinent, il faut aussi regarder les pires cas et le nombre de deadlines dépassées, ce que nous recherchons c'est des algorithmes qui sont statistiquement stables et qui dépassent le moins possible le temps qui leurs est impartis, ainsi nous tendrons a privilégier un algorithme qui est moins bon en moyenne mais plus stable dans les résultats.

La machine utilisée pour faire les mesures est un Ubuntu 18.04 avec le noyau linux : 4.15.0-50-generic, un processeur Intel i5-7200U @2.50GHz avec 3072 KB de cache et 4 cœurs, avec le gouverneur en mode performances.

Les paramètres qui peuvent influencer les résultat de l'exécution sur un même dag sont :

— Le nombre de threads utilisées
— La taille des buffers, qui fait varier le temps d'un cycle ainsi le délais. Les valeurs courantes que nous étudierons sont : 1024, 512, 256, 128, 64, 32
— L'activité de la machine, les résultats ne serons pas le même avec une machine au repos, et une autre qui a une charge de travail importante comme de la compilation par exemple.

Les Configurations de DAGS étudiées

Nous allons concentrer notre analyse sur trois configurations particulières de DAGS :

— La ligne ; chaîne de tâches ou chaqu'une s'exécute l'une après l'autre, dans cette configuration la parallélisation est impossible, toute exécution est nécessairement séquentielle, ces DAGS vont nous servir de "groupe témoin" afin de comparer le surcoût de chaque méthode.

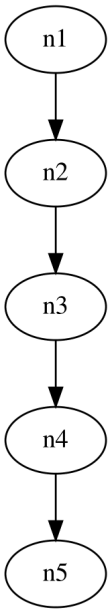


FIGURE 2 – Exemples de Ligne avec 5 nœuds

— Le Râteau ; dans cette configuration c'est plusieurs lignes qui s'unissent au bout, cette forme est celle ou la parallélisation doit être la plus efficace, et donner les meilleurs gains.

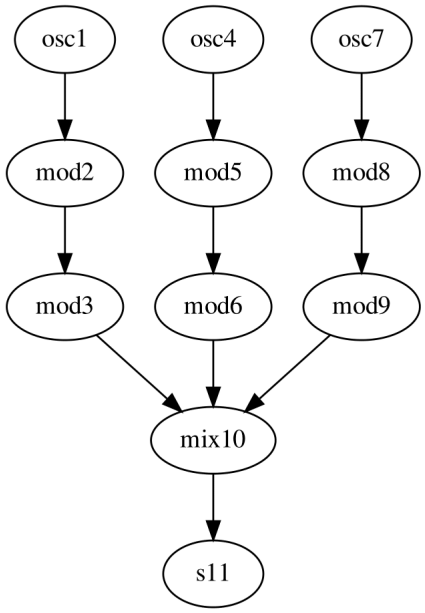


FIGURE 3 – Exemples de Râteau avec 11 nœuds

— Finalement le losange qui se trouve entre les deux, il s'agit d'un DAG qui commence linéairement puis qui va se paralléliser petit a petit pour enfin se ré-linéariser. On s'attend que l'exécution séquentielle soit plus efficace sur les petits losanges, mais que le gain de la parallélisation soit significatif sur les plus grands. Cette configuration se veut proche des cas rencontrées dans la vie réelle.

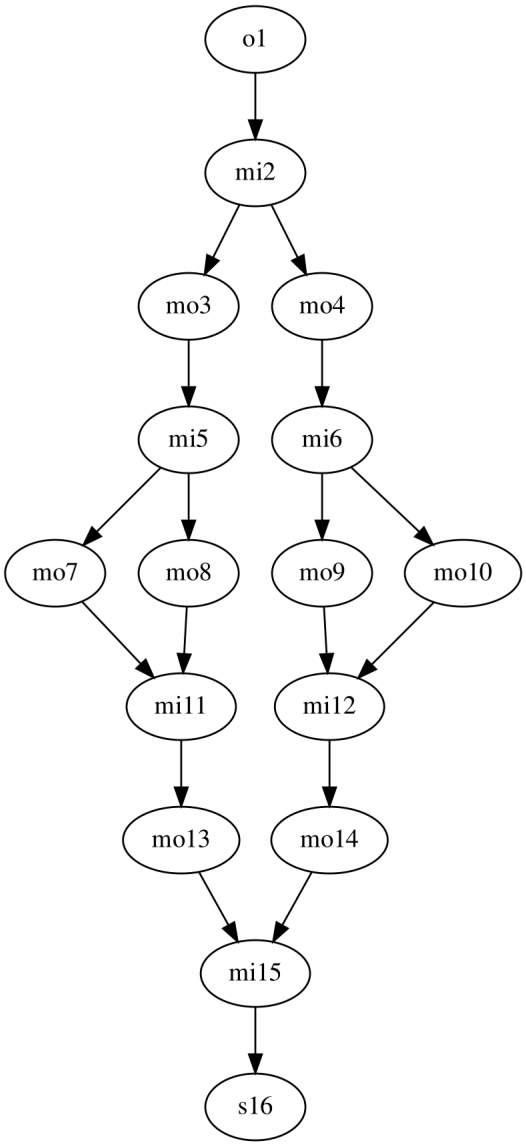
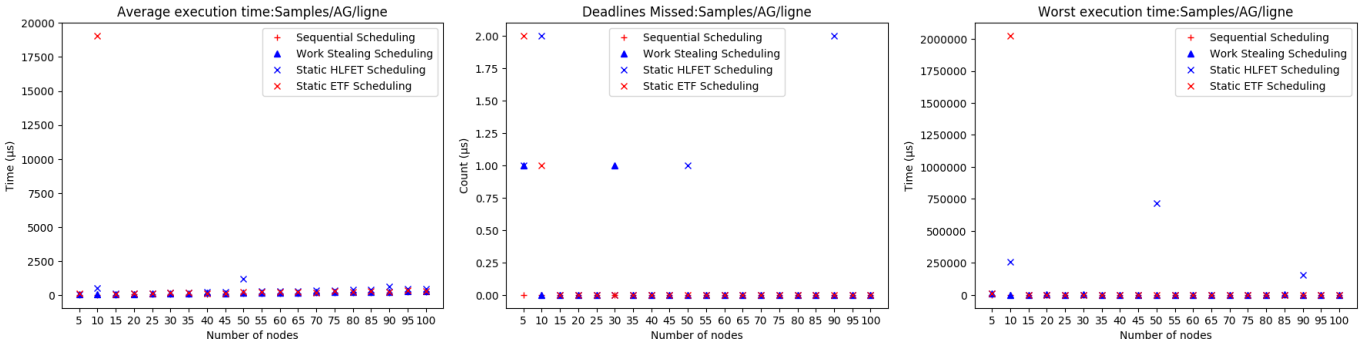


FIGURE 4 – Exemples de Losange avec 16 nœuds

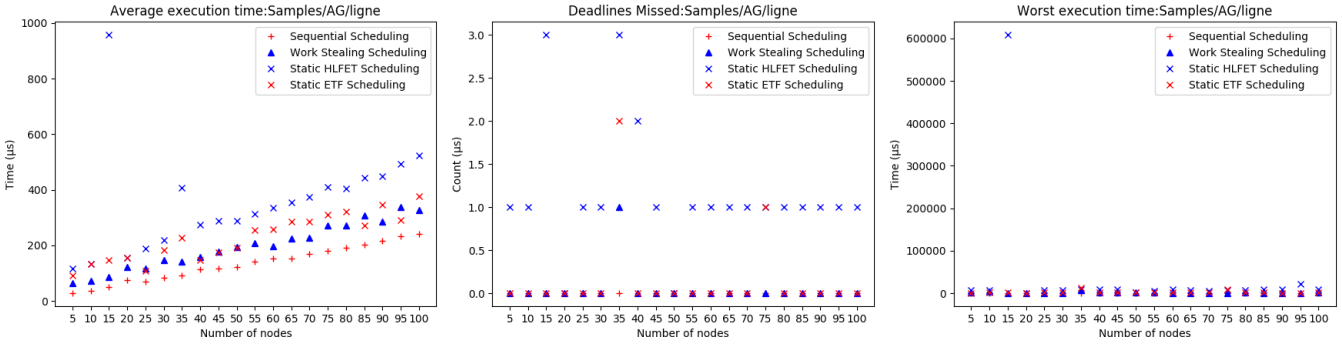
Première variable : le nombre de threads

Pour cette première série de mesures nous allons fixer la taille du buffer a **128**, et réaliser les mesures sur une machine au repos.

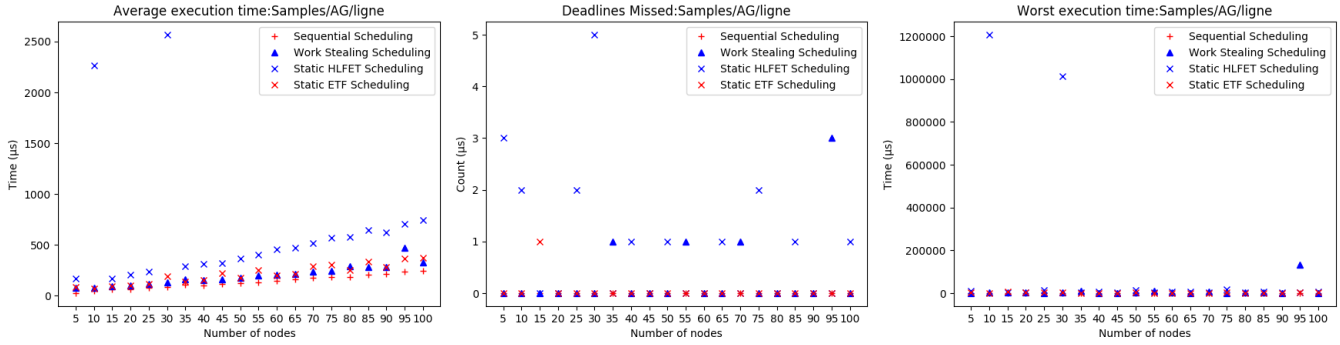
Nous commençons avec la ligne, pour pouvoir comparer le surcoût de chaque méthode [Figure 5 de la présente page].



(a) Ligne avec **2** threads



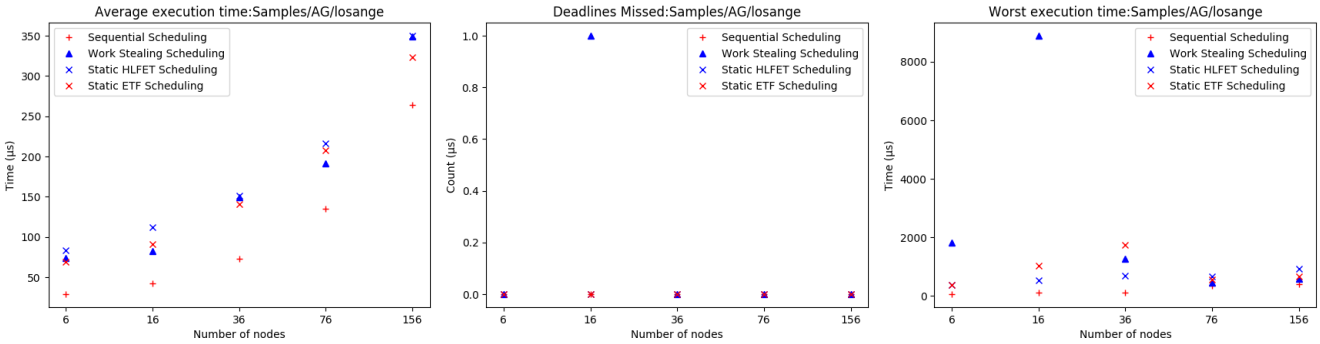
(b) Ligne avec **3** threads



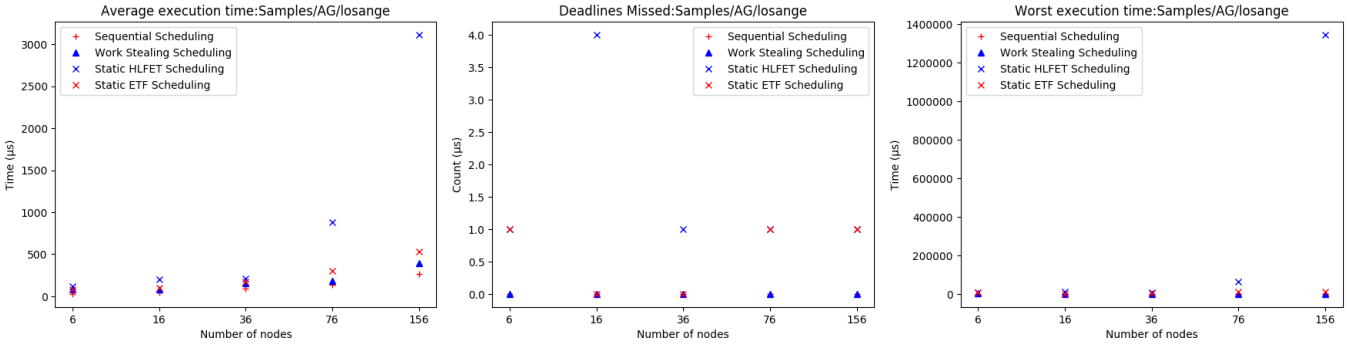
(c) Ligne avec **4** threads

FIGURE 5 – Ligne avec threads variable

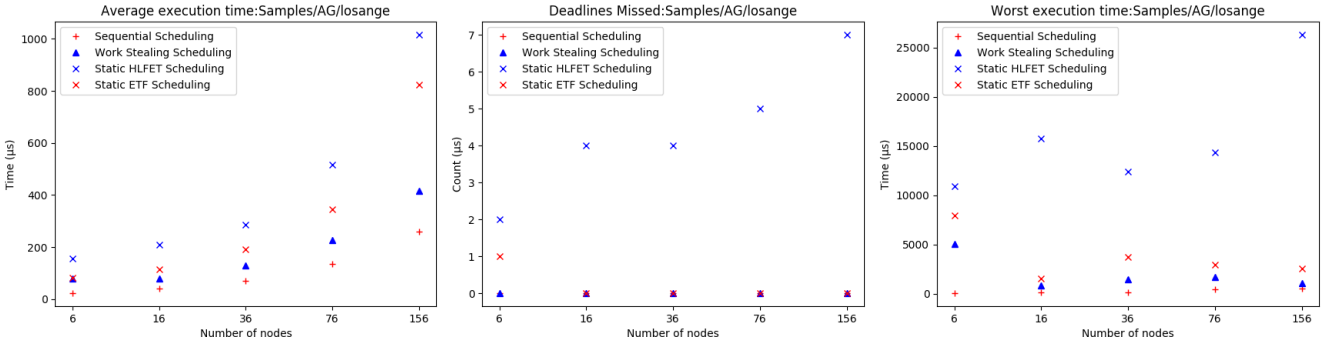
Puis avec le Losange [Figure 6 de la présente page].



(a) Losange avec **2** threads



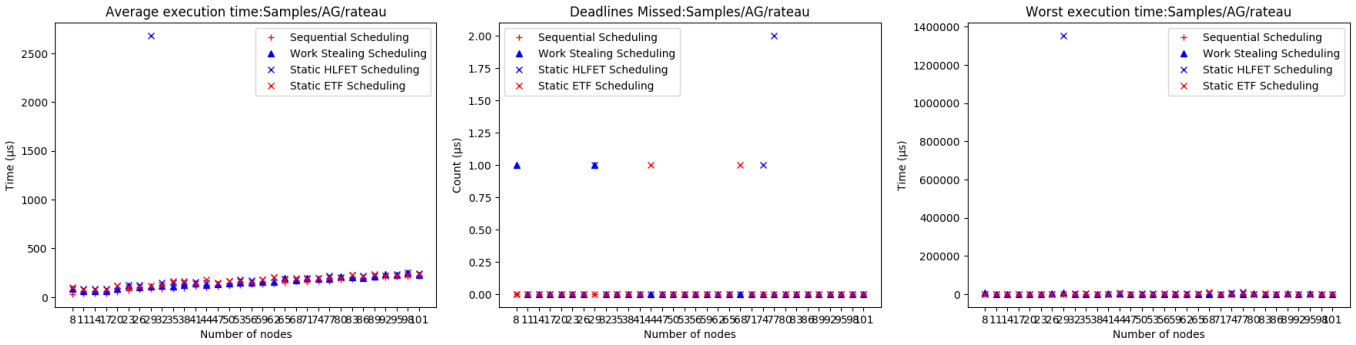
(b) Losange avec **3** threads



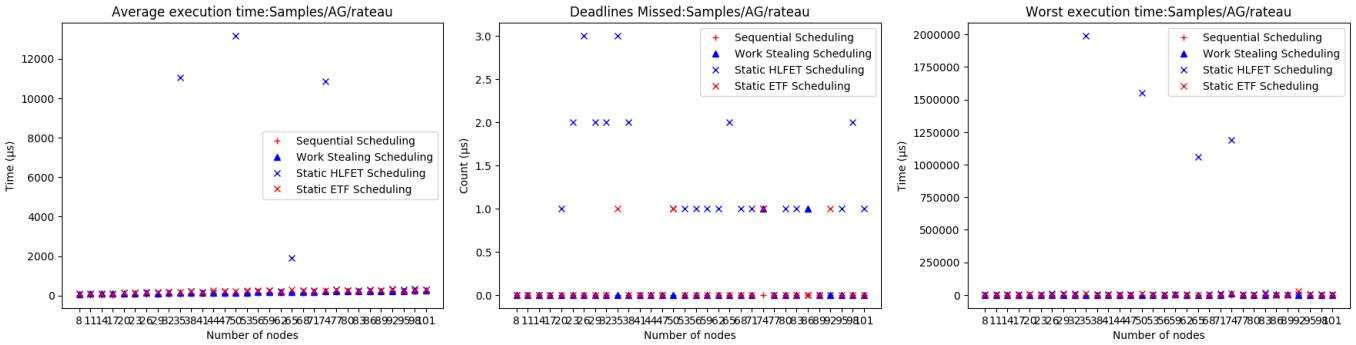
(c) Losange avec **4** threads

FIGURE 6 – Losange avec threads variable

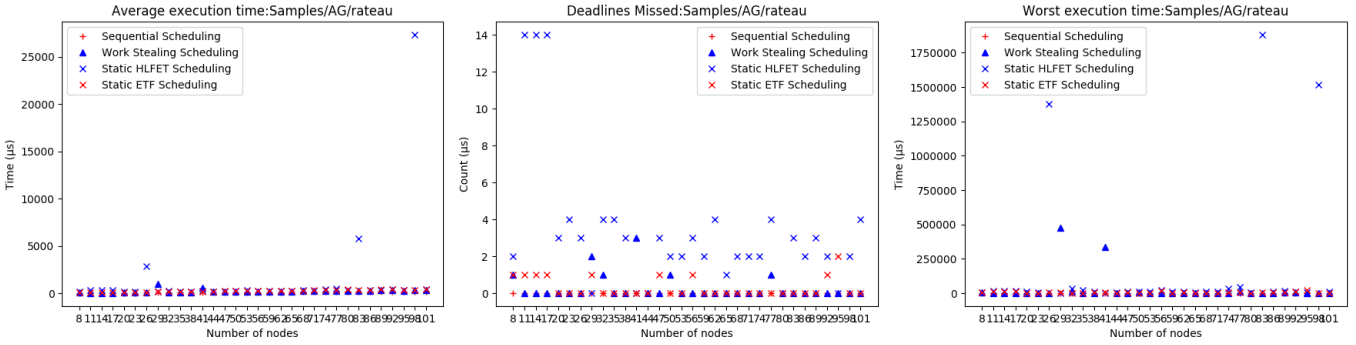
Et avec le Râteau [Figure 7 page suivante].



(a) Rateau avec **2** threads



(b) Râteau avec **3** threads



(c) Râteau avec **4** threads

FIGURE 7 – Râteau avec threads variable

Dans un Premier temps on va s'intéresser aux dépassements de délais, on peu constater dans un premier temps que le nombre de threads optimal sur cette machine est de 3 et cela quelque soit la configuration étudiée. Cela peut peut sembler contre-intuitif compte tenu qu'il y a 4 cœurs, on peut que la présence du thread de mesures impacte les performances de plus car il interrompt les threads du pool, on peut émettre l'hypothèse que le cache lui étant partagé, il est un frein a l'utilisation des 4 cœurs de la machine.

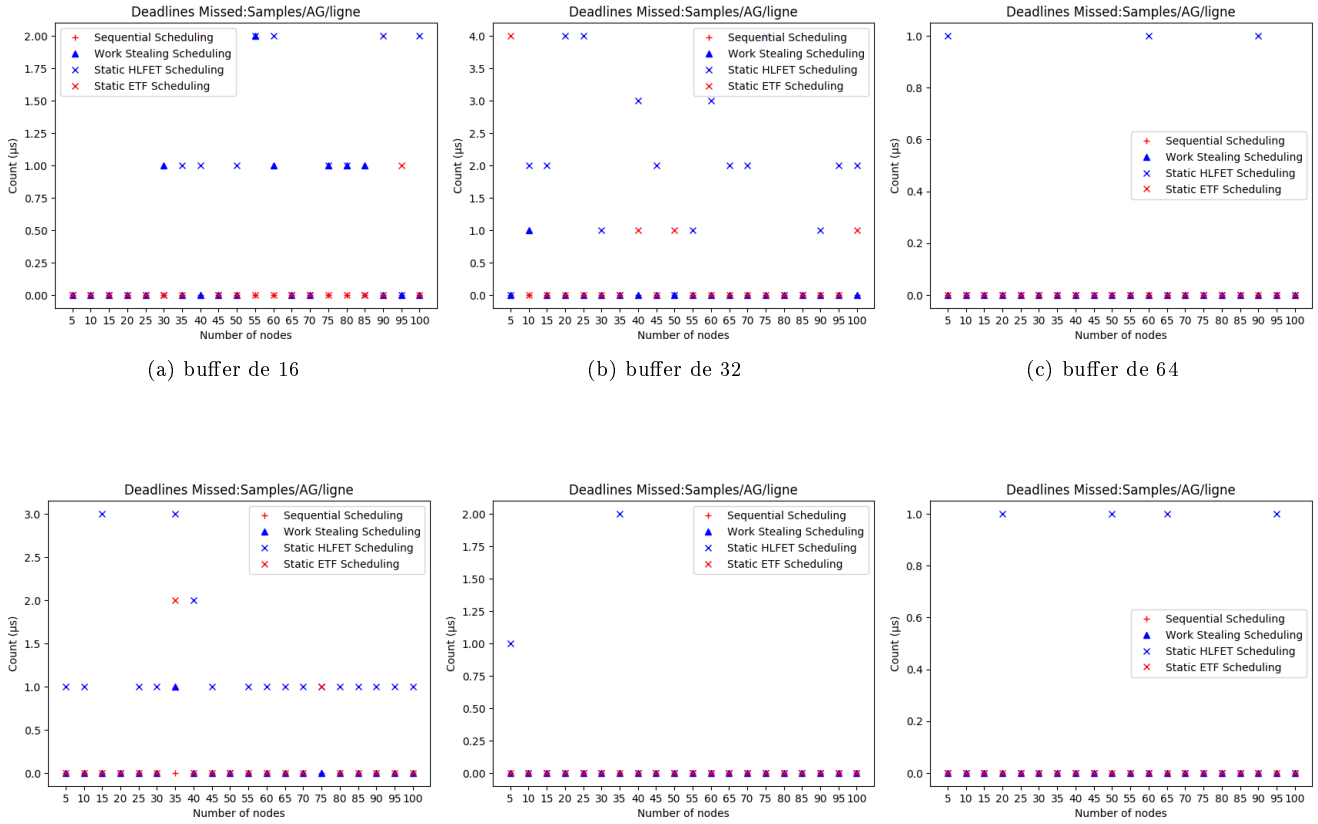
Secondement en observant les temps moyens et les pires temps pour la ligne, on peut constater que, hormis des irrégularités des méthodes HLFET et EFT, les différentes méthodes sont très proches, avec un léger surcoût de la parallélisation explicable par l'overhead nécessaire a la gestions des pool de threads et des locks.

Pour le Losange on note l'irrégularité de HLFET qui détonne par rapport aux trois autres méthodes. Dans cette configuration on voit que les 4 cœurs ne font pas perdre tant de performances que ça, juste un légère augmentation du nombre de deadlines rates.

Seconde variable : la taille du buffer

Plus la taille du buffer est petite moins le système a de latence mais plus les deadlines sont serrées et plus la machine es mise a contribution. On a deux configurations : un petit buffer entre 16 et 256, quand on cherche a avoir de la réactivité, par exemple pour des musiciens qui souhaitent avoir un retour son, pour de la retransmission en direct, et a l'opposé un plus gros buffer a partir de 512 quand on peut se permettre de ménager sa machine.

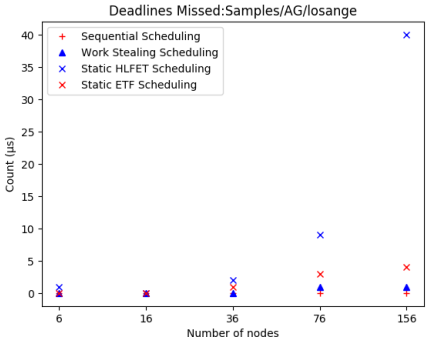
On va fixer le nombre de threads et faire varier la taille du buffer de 16, 32, 64,128, 256, et enfin 512.



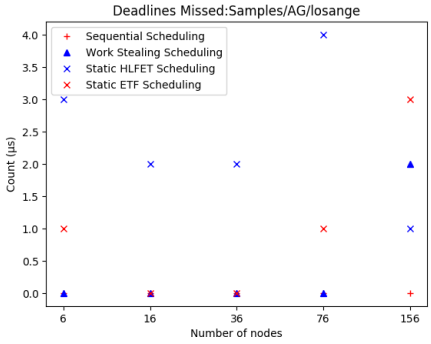
(a) buffer de 16 (b) buffer de 32 (c) buffer de 64

(d) buffer de 128 (e) buffer de 256 (f) buffer de 512

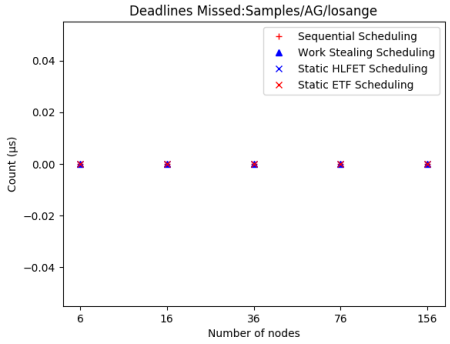
FIGURE 8 – variations pour la ligne



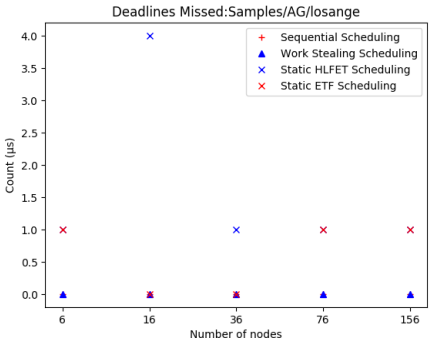
(a) buffer de 16



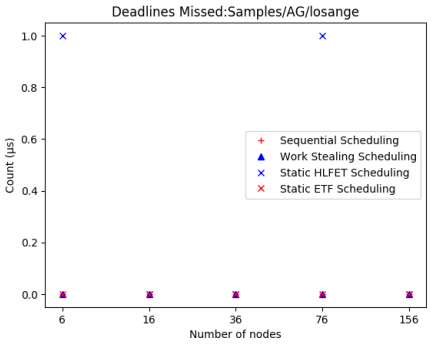
(b) buffer de 32



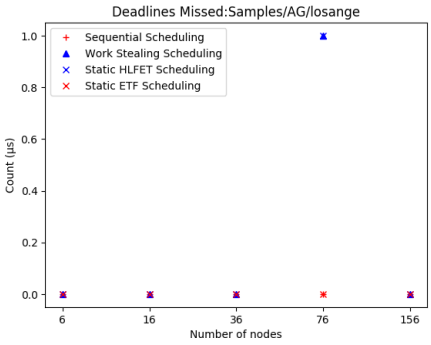
(c) buffer de 64



(d) buffer de 128

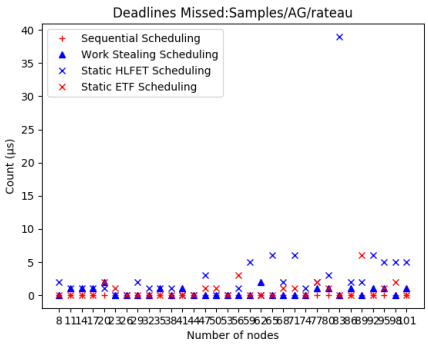


(e) buffer de 256

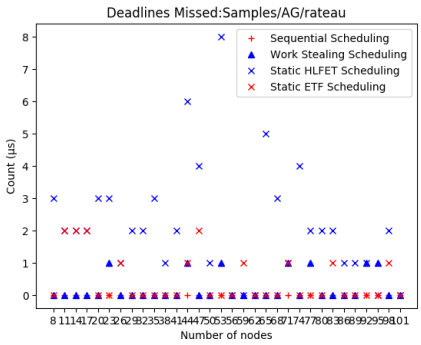


(f) buffer de 512

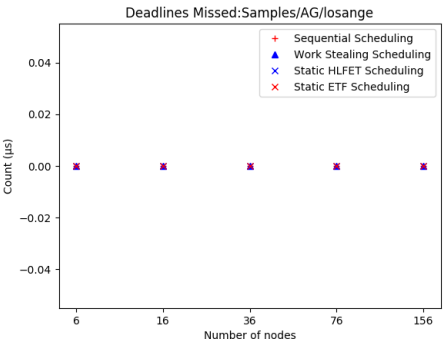
FIGURE 9 – variations pour le losange



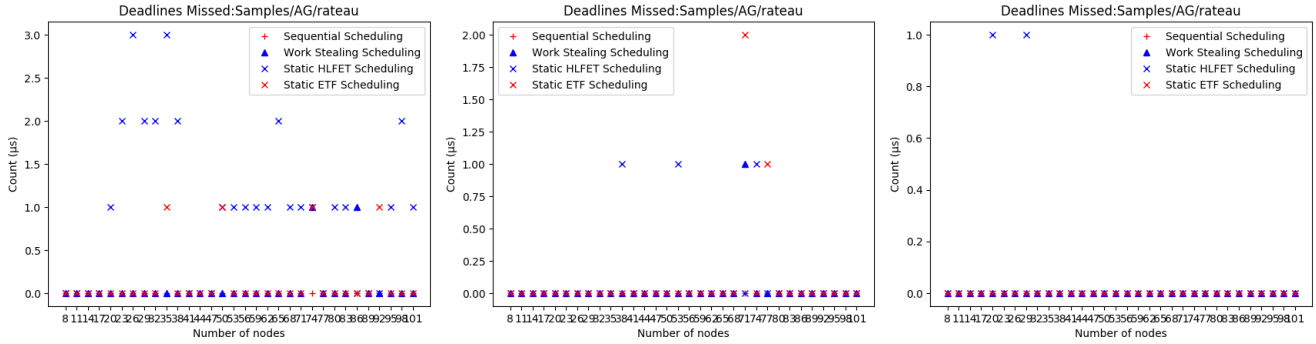
(a) buffer de 16



(b) buffer de 32



(c) buffer de 64



(d) buffer de 128 (e) buffer de 256 (f) buffer de 512

FIGURE 10 – variations pour le râteau

Dernière variable : la charge de la machine

Dans cette série de mesures on va fier le buffer a 128, le nombre de threads a 3 et on va comparer l'éxécution entre la machine au repos et la machine qui lit une vidéo en haute qualité.

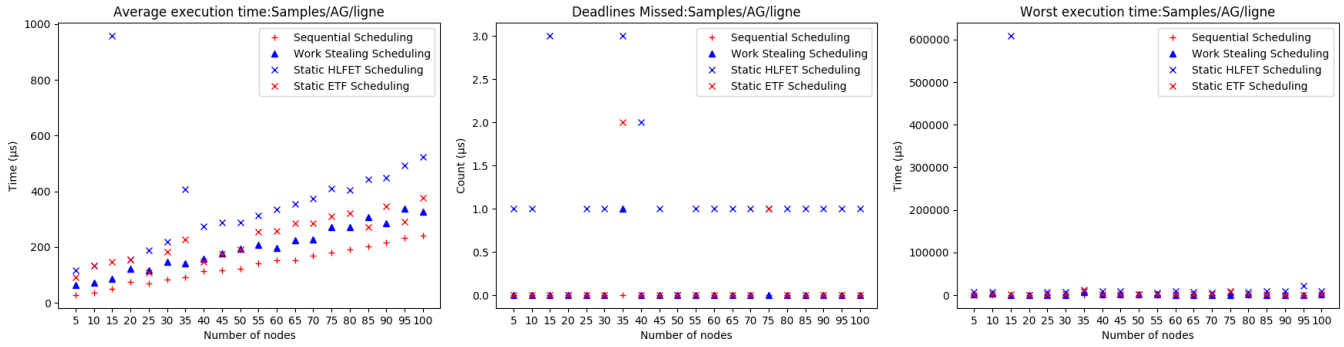


FIGURE 11 – ligne au repos

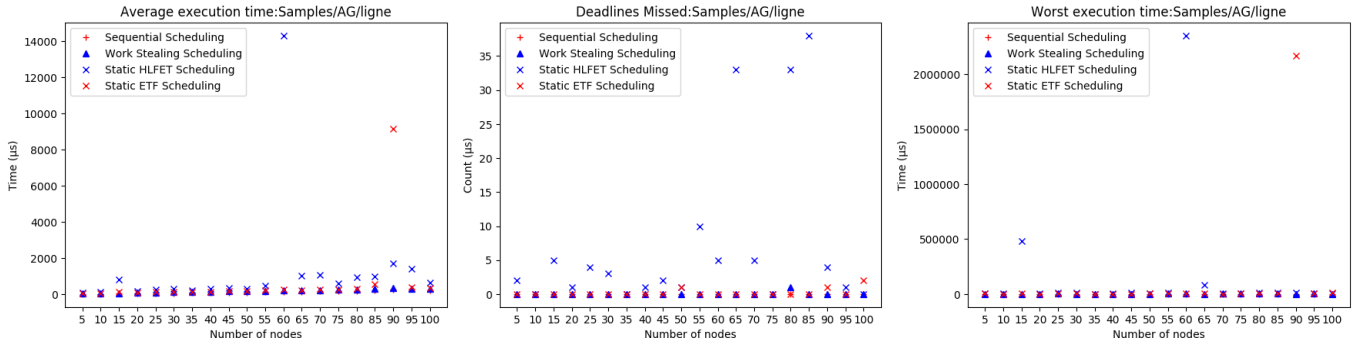


FIGURE 12 – ligne en regardant une vidéo

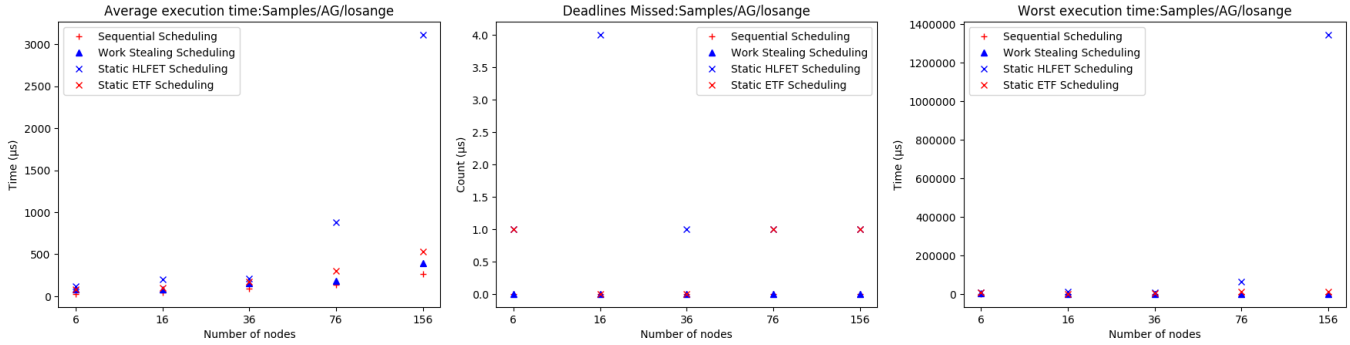


FIGURE 13 – losange au repos

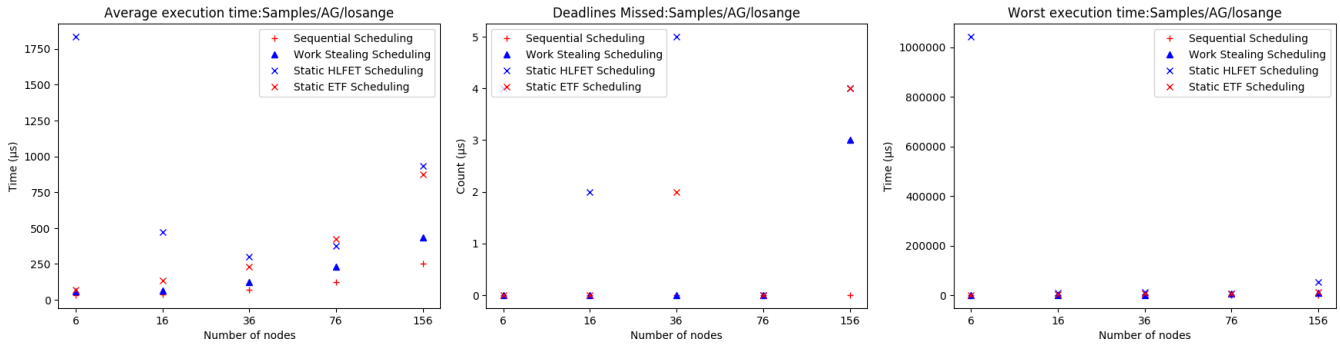


FIGURE 14 – losange en regardant une vidéo

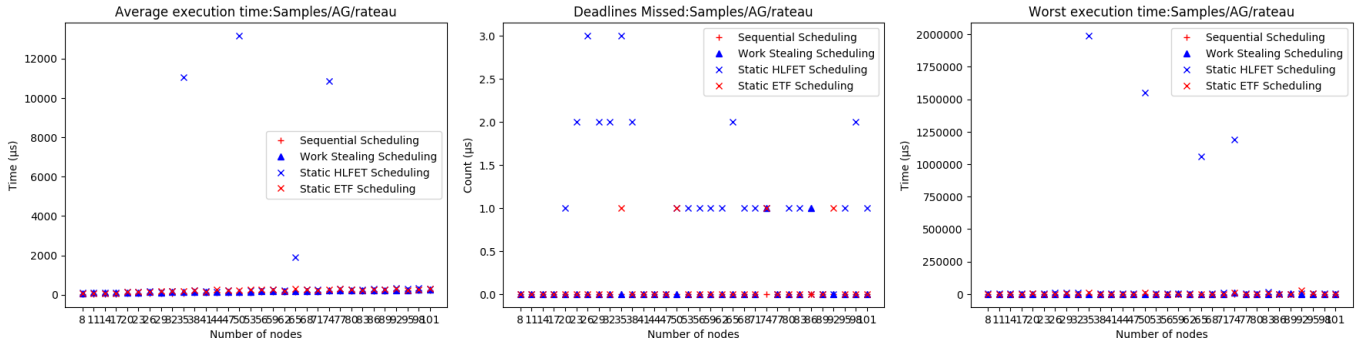


FIGURE 15 – râteau au repos

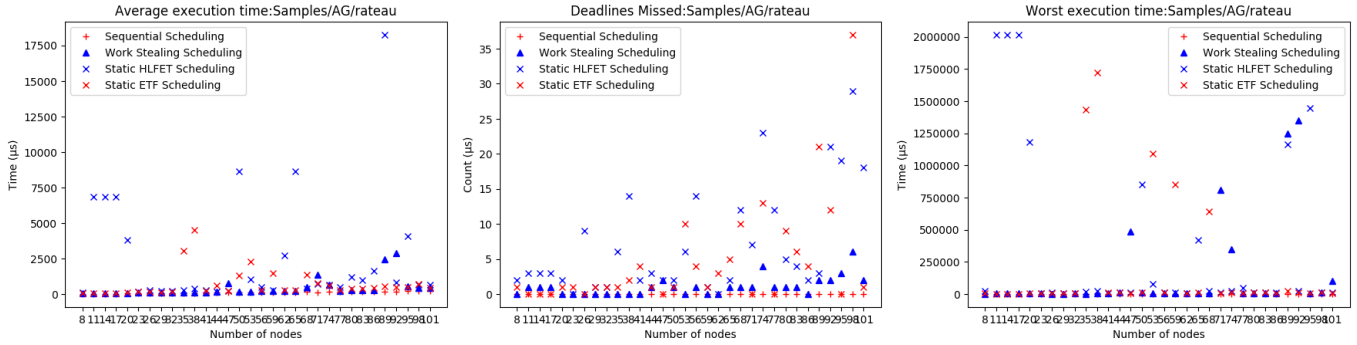


FIGURE 16 – râteau en regardant une vidéo

Le cas le plus intéressant est celui du Râteau ou on voit que lors de la lecture de la vidéo, toutes les méthodes parallèles ont plus de mal a tenir les deadlines, contrairement a l'exécution séquentielle qui elle tient la ligne avec la surcharge, on analyser cette dichotomie par le fait que la lecture d'une vidéo va monopoliser un partie non négligeable des ressources de la machine ce qui néglige une partie des gain de la parallélisation.

La stabilité des Algorithmes

Pour comparer la stabilité des Algorithmes on va faire des histogramme avec 3 threads et des buffers de 2048, a fin d'avoir des cycles plus longs et de mieux voir les variations.

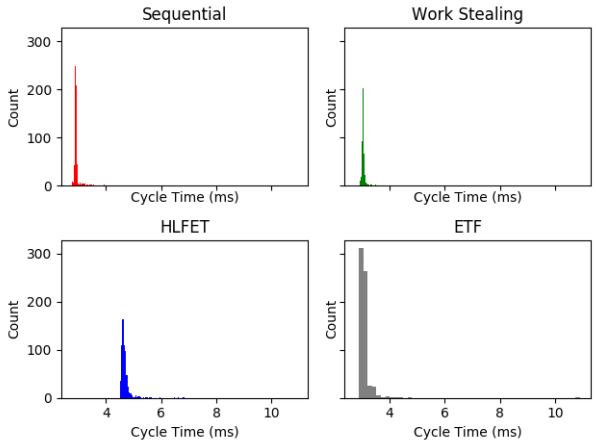


FIGURE 17 – Histogramme pour une Ligne a 100 nœuds

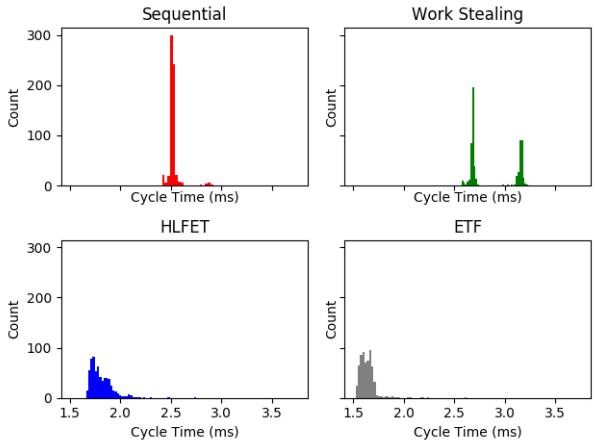


FIGURE 18 – Histogramme pour un losange a 156 nœuds

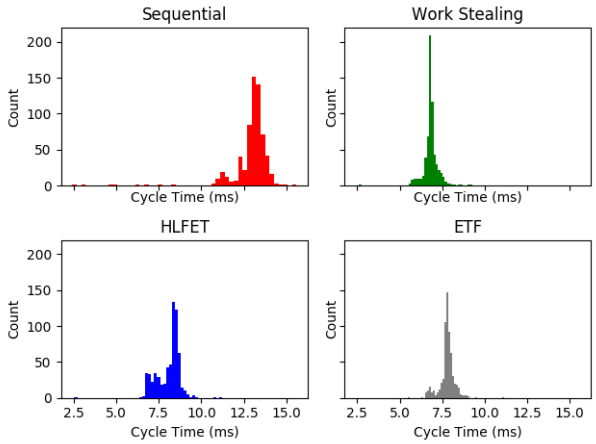


FIGURE 19 – Histogramme pour un Râteau a 101 nœuds

On peut dans un premier temps constater que sur l'histogramme de la ligne que HLFET a un overhead significatif sur les autres méthodes. Les stabilités sont proches pour les quatre méthodes.

Sur l'histogramme du losange on voit une mauvaise stabilité du vol de tâches, un peu médiocre de la part de HLFET. Les plus stables sont le calcul séquenciel en premier puis en second ETF qui est cependant notablement plus rapide.

Pour le Rateau en revanche on voir que l'exécution séquentielle n'est plus aussi stable, les méthodes parallélises gagnent en stabilité avec ETF et le vol de tache qui sont notablement plus stables.

Conclusions

On peut dans un premier temps noter la fiabilité et la constance de l'exécution séquentielle, qui dans les différents cas de figure est de loin la méthode la plus fiable et la plus efficace. Pour les autre méthodes on voit qu'HLFET est vraiment a la traîne au niveau des performances. L'algorithme ETF offre une stabilité, visible notamment dans l'histogramme du losange et du râteau, cette méthode peut être prometteuse surtout avec un soin porté a son implémentation.

Évolutions possibles

[Idée général de la partie : Critique de notre implémentation, ce qui pourrait être fait pour améliorer les résultats. Bien pour faire une conclusion "ouverte" du rapport] Pistes pour améliorer les performances de l'implémentation [Le code n'a pas été très bien optimisé en raison du temps que cela demandait] [Réduction de la quantité de locks utilisés. Les placer seulement aux endroits nécessaires et cloner le reste des données pour chaque thread] [Encore plus loin, remplacer les lock par des "traits unsafe" quand c'est possible (voir https ://doc.rust-lang.org/nomicon/concurrency.html)] [Les threads n'ont pas une priorité temps réel, on pourrait se servir de https ://github.com/padenot/audio_thread_priority ?] Fonctionnalité(s) nouvelle(s) [Utiliser MPI pour CPFD ?] [Il y a des choses meilleures que MPI ?]

Références

[1] : Yann Orlarey, Stéphane Letz, Dominique Fober, Work stealing scheduler for automatic parallelization in faust, LAC 2010

[2] : MA Kiefer, K Molitorisz, J Bieler, Parallelizing a Real-Time Audio Application—A Case Study in Multithreaded Software Engineering, Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015

[3] : YK Kwok, I Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys

[4] **https ://www.graphviz.org/** , derniere visite le 24/05/2019

[5] **https ://www.rust-lang.org/** , derniere visite le 24/05/2019

[6] **https ://llvm.org/** , derniere visite le 24/05/2019

[7] **http ://www.jackaudio.org/** , derniere visite le 24/05/2019

[8] **https ://docs.rs/crossbeam/0.7.1/crossbeam/** , derniere visite le 24/05/2019

[9] **https ://docs.rs/core_affinity/0.5.9/core_affinity/** , derniere visite le 24/05/2019

[10] **https ://criterion.readthedocs.io/en/master/** , derniere visite le 24/05/2019

[11] **https ://pest.rs/** , derniere visite le 24/05/2019

1 Soit *candidate* le CPN d'entrée.
2 **Répéter :**
3 Soit *P_SET*, l'ensemble des processeurs comportant les parents de *candidate*, plus un processeur inutilisé.
4 **Pour chaque** *P* dans *P_SET*, faire :
5 **(a)** Calculer *ST* : le temps de départ de *candidate* sur *P*.
6 **(b)** Soit m, un éventuel parent de *candidate* qui n'est pas ordonnancé sur *P* et dont le message pour *candidate* a le temps d'arrivée le plus tardif.
7 **(c)** Essayer de dupliquer m sur le créneau d'inactivité de *P* le plus précoce.
8 **Si** la duplication réussi et que cela diminue *ST*
9 alors mettre à jour *ST*. Faire en sorte qu'un nouveau candidate soit m et retourner à l'étape **a**. .
10 **Sinon**
11 si la duplication échoue, alors rendre le contrôle pour examiner un autre parent du candidate précédent.
12 Ordonnancer candidate sur le processeur *P'* qui lui permet de commencer le plus tôt et faire les duplications requises.
13 Soit *candidate*, le CPN suivant.
14 Répéter le processus de l'étape **3**. à **6**. pour chaque OBN avec *P_SET* contenant tous les processeurs utilisés, plus un processeur inutilisé. Les OBN sont ordonnancés dans l'ordre topologique.
15 **Fin si**
16 **Jusqu'à ce que** tous les CPN soient ordonnancés.

Algorithme 2 : Ordonnancement CPFD